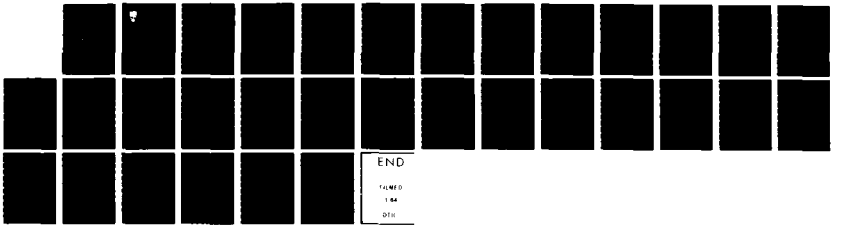
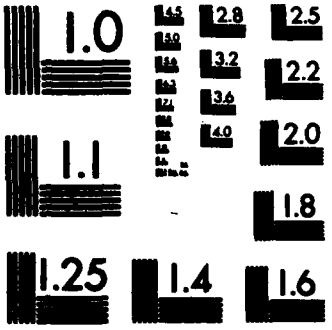
AD-A135 513 KERNEL AND SYSTEM PROCEDURES IN FLEX(U) ROYAL SIGNALS 1/1
AND RADAR ESTABLISHMENT MALVERN (ENGLAND)
I F CURRIE ET AL. AUG 83 RSRE-MR-3626 DRIC-BR-89750

UNCLASSIFIED F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

UNLIMITED

RSRE
MEMORANDUM No. 3626

ROYAL SIGNALS & RADAR ESTABLISHMENT

KERNEL AND SYSTEM PROCEDURES IN FLEX

Author: I F Currie, J M Foster
P W Edwards

PROCUREMENT EXECUTIVE,
MINISTRY OF DEFENCE,
RSRE MALVERN,
WORCS.

UNLIMITED

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Memorandum 3626

TITLE: KERNEL AND SYSTEM PROCEDURES IN FLEX

AUTHOR: I F Currie, J M Foster, P W Edwards

DATE: August 1983

SUMMARY

This Memorandum describes the basic Kernel and System procedures on which the operating system for the Flex computer is based. These are the low level procedures which are used to implement the compilers, file-store, command interpreters, etc. on Flex.

Kernel and System procedures in Flex

Contents

1. Introduction

The Kernel of Flex is a set of procedures which do the resource allocation and raw peripheral transfers of the Flex system. Generally these procedures run in privileged mode (see Flex Firmware [1]); in fact, one could almost define Kernel as being that set of procedures which run in privileged mode.

Some of the procedures in Kernel are available directly to the general user; these form the Kernel interface. To the user, all procedures are treated in the same manner, so he should find no difference between a Kernel procedure and any other. In common with all Flex procedures, the pointer which is a Kernel procedure can only be passed around and called - one cannot use the pointer to dig into information which the procedure uses. To this extent then, the Kernel interface should be regarded as an extension of the firmware in that the user is completely free to use any of the non-privileged instructions and any of the interface procedures in any manner.

The exact definition of a what is meant by a system procedure is much less clear. The procedures which I call System procedures are all written in non-privileged mode. They form the interface between the user and a particular operating system written on top of the Kernel. This operating system is sometimes called Curt, although the name should more properly be applied to the command interpreter of the Flex operating system [2]. The charactertistics of this operating system are such that the dividing line between system and non-system procedures is largely dependent on the observer; a user could construct a new system in Flex which uses as much or as little of the current system as he required without a great deal of trouble. Thus, the choice of the particular procedures to be described as System procedures is largely subjective. My usual criterion is to ask whether I can give a program text which describes it completely, taking other Kernel and System procedures as being defined. If the answer is no, then the procedure is an System procedure. This usually arises from the fact that there is some non-textual value bound into the procedure, for example a dictionary held in file-store.

The procedures described in chapters 2-5 are all Kernel procedures with most of the rest being simply non-privileged System procedures. They all happen to have been written in Flex Algol 68 and so the descriptions of the interface procedures in the following chapters are generally expressed in Algol 68. However this does not imply that they can only be called from Algol 68 programs or indeed that Flex is an Algol 68 machine. The underlying structure of Flex is much more powerful than any hypothetical Algol 68 "machine". The total relaxation of all of the Algol 68 scope restrictions in the Flex compiler is one indication of this.

2. Virtual machines and Processes

2.1 General

A Virtual Machine on Flex is initiated by the reception of an WHITE Control/A from a vdu. This results in the construction of procedures and data structures particular to the Virtual Machine. Each Virtual Machine gets a time-slice, sharing the total time with other Machines in a round-robin. Communication between Virtual Machines is usually done via the file-store.

Each Virtual Machine has a set of processes which (given that they are not held up for other reasons) will be run in a local round robin using process time-slices, regarding the Virtual Machine time-slices as though they were contiguous. Any process which does not complete its process time-slice through being held by a peripheral transfer, will be run for a time equal to its unexpired time-slice as soon as the transfer is completed.

The set of processes of a Virtual Machine initially consists of :

1. The normal control process , which will usually be the one which runs Curt and normal programs. This is the principal process of the Machine.
2. The break-in process which initially is in a loop demanding to read the BREAK IN LINE on the vdu. If and when this read is completed by sending the break-in line, process 1 will be failed.

The Kernel interface procedures given here allow one to create, identify and synchronise processes.

2.2 PROC make_process = PROC (PROC VOID)VOID:

Make_process is a procedure which will create a new process in the Virtual Machine. A call of make_process delivers a procedure called the soft interrupt for this new process. In order to run a PROC VOID f in the new process, one calls the soft_interrupt with f as a parameter. A subsequent call of the soft_interrupt with another PROC VOID g will result in a FAIL(0, g) if f is not completed in the new process; otherwise g will run in the new process.

2.3 PROC own_si = (PROC VOID action)VOID:

The procedure own_si is the soft_interrupt for the principal process of the current Virtual Machine. This would usually be used to fail this process from other processes.

2.4 PROC break_si = (PROC VOID breakin_action)VOID:

The procedure break_si is the soft_interrupt for the break-in process of the Virtual Machine. The initial procedure running in the break-in process is written so that failures are trapped and, in particular, the FAIL(0, g) produced by break_si(g) will result in g

being called from within the break_in process. This may be used to allow the user to create his own break-in action.

2.5 PROC make_sema = (INT n)SEMA:
Where SEMA = PROC(BOOL)VOID;

The value of the parameter, n, of a call of make_sema is the initial value of a new semaphore given by the answer to the call. The operation of downing (securing) the semaphore is performed by applying a FALSE parameter to the SEMA; raising (releasing) the semaphore by applying a TRUE parameter. The usual definition of a semaphore applies; the value of a semaphore is non-negative and any process which tries to down a zero semaphore is suspended until the semaphore is non-zero.

2.6 PROC timed_wait = (INT secs)VOID:

The call, timed_wait(s) holds up the action of current process for at least s seconds, running thereafter when there are no other runnable processes.

2.7 PROC own_time = LONG INT:

The value delivered by own_time is the time in milli-seconds that the current process has been running. (The time of day is given by an instruction, opcode 35).

2.8 VECTOR[]CHAR date

The current date given in the form day/month/year eg. 10/1/83

3. File stores

3.1 General

Usually the user will only be aware of one file-store, entirely resident on disc. His knowledge is defined entirely by the disc kernel procedures and disc pointers that he possesses. These disc procedures are arranged so that (except in one carefully controlled case) no overwriting of data occurs - one can only read data from an existing disc pointer or write a chunk of data to create a new block on disc pointed at by a new disc pointer.

As explained in [1], the main memory representation of a disc pointer is a locked pointer to a 4-word block. The key for unlocking this pointer is the same for all pointers belonging to the same file-store. Disc pointers can also exist in their own file-store where their representation is 4 bytes. The mediation between the two representations is done by the disc reading procedures in such a way so that if a disc pointer to a file-store block is read in two different places the main memory representation will be the same locked pointer in both cases. This implies that it is relatively easy to alias file-store blocks with main memory blocks to facilitate sharing of common blocks. The correspondences and alias information are tied together with shaky pointers so that the memory does not become clogged up with disc pointers and their aliases.

The only exception to the no overwriting rule is the use of disc references. A disc reference is essentially a variable on disc which can contain a disc pointer. This variable can be read using d_to_b and can be assigned to (with caveats) using to_dr.

Since most writing to the disc will increase the size of the filestore, the filestore is periodically garbage-collected. This is done by an off-line program and, except for the fact that the filestore is inaccessible during garbage-collection, the user is usually unaware of its action, except for its influence on shaky pointers as described in 3.5.

In the following procedures a DISCPTR or DISCREF is formally an Algol 68 INT, since no Algol 68 mode constructor can adequately describe a disc pointer.

3.2 Disc writers

```
3.2.1  PROC npb_to_d   = (VECTOR[]XTYPE data) DISCPTR:
         where  XTYPE is FLAT
                      or VECTOR[]FLAT,
           where FLAT  is INT, REAL, CHAR ...,
                       or STRUCT(FLAT....,...),
                       or UNION(FLAT,....).
```

This procedure writes data to a block on disc delivering a disc

pointer to that block where data contains no pointers. The mode of the parameter allows one to gather together data of diverse non-pointer modes.

3.2.2 PROC pb_to_d = (VECTOR[]XTYPE data) DISCPTR:

This procedure writes data to a block on disc delivering a disc pointer to that block where data may contain disc pointers. Once again the data written is gathered together; this time however each item written must be multiples of words.

3.2.3 PROC c_to_d = (INT ws,REF VECTOR[]INT consts,
REF VECTOR[]CHAR code,BOOL proc)DISCPTR:

This procedure creates a code-block on disc, where ws is the work-space size required by the code. The constants of the code_block, which may contain other disc pointers are given in consts. A code_block disc pointer is delivered if proc is FALSE; otherwise a disc pointer to a procedure with no non-locals is delivered.

3.2.4 PROC p_to_d = (DISCPTR cb,VECTOR[]XTYPE nls)DISCPTR:

This procedure creates a procedure on disc,with code-block given by the code-block disc pointer cb (produced by c_to_d) and non_locals given by nls (may contain disc pointers and must be in word multiples). A disc pointer to the procedure is delivered.

3.3 Disc readers

3.3.1 PROC d_to_b = (DISCPTR d)PTR:
where PTR = INT;

This procedure reads the disc block corresponding to disc pointer d, into a main memory block and a pointer to this block is delivered as answer. The type of main memory block produced depends on the type of the disc pointer d which in turn depends on how it was produced.

Thus:
if d had been produced by npb_to_d the block type is 3 (non-pointer);
if d had been produced by pb_to_d the block type is 4 (normal);
if d had been produced by c_to_d the block type is 2 (code-block);
if d had been produced by p_to_d the block type is 5 (closure);
if d is a disc reference then answer is the contents of the disc reference, which is usually a disc pointer (type 6 block).

In all cases the pointer delivered is such that the contents of the block cannot be altered i.e. in the first three cases above (types 2, 3 and 4) the pointer is locked. This means that this same block can be given as an answer to any call of d_to_b with the same parameter and this block can be shared between many users and programs. Since all code_blocks are read into memory using d_to_b, the saving in both disc access and main memory usage is quite significant.

3.3.2 PROC from_disc = (DISCPTR d,VECTOR[]REF XTYPE start)REF YTYPE:
Where REF YTYPE can be coerced to {REF} FLAT,
or to {REF} VECTOR[]FLAT.

This procedure reads the data from the block pointed at by d. This disc pointer must have been produced by some call of npb_to_d or pb_to_d and the data read is identical to that given by the parameter of this call. The data is assigned to the variables given by the parameter, start, in order starting from the beginning of the data and any remaining data is delivered as the answer. Usually from_disc will be called in such a context so that this remainder (of mode REF YTPE) will be coerced to some vector. Clearly, if disc pointers are being read into start, then the corresponding variables in start must occur in multiples of words.

The procedure from_disc can be used as an effective inverse to either npb_to_d or pb_to_d, providing a scatter-gather pair.

3.4 Disc reference procedures

Disc references can be read using d_to_b (3.3.1).

3.4.1 PROC to_dr = (DISCREF dr,INT old,new)VOID:

This procedure assigns new (usually a disc pointer) to the disc reference dr, provided that the old value contained in dr was old (integer or disc pointer). If the old value was not old then the procedure fails. Since to_dr acts on the file store as an elementary operation, this check allows a fairly simple method of resolving difficulties caused by simultaneous altering of dictionaries pointed at by disc references.

3.4.2 PROC new_dr = DISCREF:

This procedure creates a new disc reference (containing integer 0) and delivers it as an answer. This procedure is hidden from most users but is included here for completness.

3.5 Shaky disc pointers

Disc pointers produced by npb_to_d or pb_to_d can exist in a "shaky" form. That is to say, so long as a normal or "firm" version of a pointer exists somewhere in the filestore then this firm version can always be recovered from a shaky version of the pointer by application of the procedure firm_dptr below. If there is a shaky disc pointer with no corresponding firm version in the filestore then the shaky pointer will be replaced by 0 (and any associated space recovered) when the filestore is garbage-collected.

3.5.1 PROC firm_dptr = (DISCPTR shaky) DISCPTR:

This procedure delivers the firm version of its shaky disc pointer parameter.

3.5.2 PROC shake_dptr = (DISCPTR ptr) DISCPTR:

This procedure delivers the shaky version of the parameter ptr which was produced by npb_to_d or pb_to_d.

4. Vdus

4.1 General

When WHITE Control/A is typed on a vdu, Kernel creates a specific vdu procedure to go with the Virtual Machine (see 2.1). This procedure is now the only one capable of communicating with the particular vdu and will remain valid until WHITE Control/A is typed again on that vdu.Thus a Virtual Machine usually has sole possesion of a physical vdu simply by being the only one which has the correct vdu procedure.

The Logica vdus have the capacity to display characters in various forms eg reversed video, flashing, underlined etc. An underlined character is indicated by adding 128 to the 0 - 127 ISO representation of the character sent to the vdu. Other forms are indicated by the parameters of the procedure vdu.

Each call of the vdu procedure is a single interaction consisting of writing some lines (possibly empty) to an area of the screen and then, optionally, waiting for a control key to be pressed to indicate that the lines on some area of the screen are to be delivered as the answer to the procedure. Several interactions may be in progress simultaneously to the same vdu arising from calls of the vdu procedure in different processes. The NEXTREAD sequence on the vdu keyboard will cycle the cursor through the various read areas to allow any of the extant interactions to be completed. The most obvious example of this is the break-in process, which is usually held up waiting for the top line of the screen to be sent.

```
4.2  PROC vdu = (VECTOR[]INT pre,VECTOR[]TEXT mess,VECTOR[]INT post,
                 VECTOR[]REF INT ans)REF VECTOR[]LINE:
          Where TEXT = UNION(REF VECTOR[]LINE, VECTOR[]CHAR,
                             VECTOR[]VECTOR[]CHAR),
          and   LINE = REF VECTOR[]CHAR;
```

This procedure sends the lines defined by mess to an area of the screen called the write area which is defined by the vector pre. Each element of the vector mess starts on a new line. If a component of mess is a REF VECTOR [] LINE then each of its elements will start on a new line. The components of a VECTOR [] VECTOR [] CHAR are concatenated to form one line.

If post is an empty vector, then this is a write-only interaction and NIL is delivered as the answer. Otherwise post defines a read area on the screen the lines of which will be delivered as the answer to vdu when the interaction is completed by keyboard action. The variables in ans will then show how the the interaction was completed.

The vector pre must have upper bound 7, and the significance of each of its elements is as follows:

pre[1] : any or all of these bits may be present :-
2 - Lines after the last written will not be cleared.
4 - The cursor will not be shown for the read unless it is invoked by next_read on the vdu
8 - There will be a left margin during the read part of the exchange into which the cursor will not move. Only the characters to the right of this will be returned. The position is set by post[4]
16- The area will be read without interacting with the user.

pre[2] : The line width in characters [1 : 160].

pre[3] : Absolute line number of first lin. of write area where the top line of the screen is 0, the bottom one 23.

pre[4] : Number of lines of required width in write area.

pre[5] : Relative line number (0 is the first line) within current write area onto which the first LINE will be written. Unless 2 is present in pre[1] the remaining lines in the area will be cleared. pre[5] may lie between -23 and 46. If it is outside the write area the data in the write area will be scrolled up or down so that the first line is just inside the area. If enough lines are written to go past the bottom of the area the data in the write area will be scrolled up.

pre[6] : Character position [0 : pre[2]-1] in line from which writing will start.

pre[7] : "Colour" used. Any or all of the following bits may be present :-
1 - Blink
2 - Reverse video
4 - Half intensity
16- Invisible

The vector post must have upper bound 0 or 5, and in the latter case, the significance of each of its elements is as follows:

post[1] : Absolute line number of first line of read area [0 : 23].

post[2] : Number of lines of required width in read area.

post[3] : Relative line number [0 : post[2]-1] within read area for initial position of cursor.

post[4] : Character position [0 : pre[2]-1] for initial position of cursor.

post[5] : "Colour" to be used for read area. As for write.

The vector ans must have upper bound <= 4 , and the significance of each element, provided it exists and is not NIL, is as follows:

ans[1] : A codification of the keystroke(s) which sent the read data
152 - DUPL
153 - DEL LINE
148 - INS LAST
150 - INS LINE
145 - up arrow
147 - south west arrow
139 - down arrow
176 - VOID
177 - GO IN
179 - RESULT
181 - DO
<128 - given by WHITE/x and ans[1] is ascii code for x

ans[2] : Multiplier number; will be 1 unless the *10 key has been used immediatly before the sending keystroke(s), in which case it will be a multiple of 10 < 230 minus the number of times the operation was performed on the screen before needing to send lines. If it is >230 then the *MAX key was pressed.

ans[3] : Relative line number [0: post[2]-1] of final position of cursor.

ans[4] : Character number [0: pre[2]-1] of final position of cursor

The lines returned depend on the key which was pressed to send the data back.

Let q = min(post[2], ans[2])
r = min(post[2] - ans[3], ans[2])
VOID, GO IN, RESULT, DO send back the whole read area.
Up arrow sends back the bottom q lines.
Down arrow and south west arrow send back the top q lines.
INS LAST sends back the bottom r lines.
DEL LINE and DUPL send back r lines starting from line ans[3]
INS LINE sends back the bottom r lines if ans[4]=0, the bottom post[2]-ans[3] lines if ans[4]/=0.
WHITE/x sends back no lines.

The definition of the lines which are returned may appear complicated; however, it can be summarised by saying that the minimum number of lines are sent, compatible with the implementation of the nominal action of key pressed.

5. Exceptions

As described in the Flex Firmware [1], a exception in D-state provokes the call of an otherwise inaccessible Kernel procedure, failure, to record and pass on information about the exception. The failure procedure is called in place of the procedure call which contained the instruction which raised the exception and it generates a structure of mode F where:

```
MODE F = STRUCT(PTR locals,INT pc,sf, PTR code_block, EP ep)
```

where the fields are:

- locals is a pointer to the workspace in which the exception occurred.
- pc is byte displacement from the start of code of the instruction in code_block which was in error.
- sf is byte displacement from the start of locals of the stack front when the exception occurred.
- code_block is a pointer to the code block in which the exception occurred.
- ep is the error pair corresponding to the error.

Since failure does a Exit-fail instruction (Op code 69) with a ref to this structure in U, the mode EP could be REF F. The net effect is that a chain of mode REF F is formed by successive failures through various procedures until an instruction is encountered which can cope with the illegal value produced. This is usually the unite-illegal instruction (opcode 165) which allows one to access the characteristic word pair of the illegality (in this case the REF F). This chain may then be analysed by procedures such as diagnose to indicate error positions in the code and the values of locals in the chain of failing procedure workspaces.

6. Loading and assembling program

Compilers for Flex use the primitive disc writing procedures to create filestore codeblocks and procedures. Thus, most of the code-blocks used in Flex are formed by creating a disc code-block using c_to_d. Of course, the code can only be obeyed when this code-block is loaded from disc into main memory. This usually takes place when a procedure made from this code-block is called - an interrupt occurs in the middle of the calling sequence and the interrupt procedure then called will replace the disc code-block pointer with a main memory pointer using d_to_b (see [1]). This is entirely invisible to the user and hence the distinction between disc and main store code-blocks tends to be rather academic.

The basic unit of programming in Flex is the procedure and a Flex compiler produces a filestore procedure corresponding to the source text as a part of its result. This procedure is one which will deliver the interface values defined in the source text as its result, eg those values corresponding to the KEEPlist in an Algol68 module or the "specification" part of an Ada compilation unit. In the course of evaluating the procedure, a value from another module or compilation unit USEd by the former is derived simply by calling its corresponding procedure. Thus the filestore procedure corresponding to a program text contains (usually in its non-locals) means of deriving the filestore procedure corresponding to each of the external modules used by that program text. Usually, the external filestore procedure is not itself in the non-locals, but rather a kind of a reference to the procedure called a Module.

The reason for a Module to be a reference to the procedure rather than the procedure itself lies in the requirement that one should be able to recompile and reconstruct program units without invalidating other units which use them. This implies some degree of assignability. The details of whether or not one can change a Module belongs more properly to a description of the language and compiler; however the kernel of Flex does define a language independent mechanism for the construction of programs in this modular fashion such that one can ensure their consistency at least at the modal level.

6.1 Modules

A Module in Flex is a three word object:

```
MODE MODULE = STRUCT( DEREF deref,ASSIGN assign, SPEC spec)
```

This structure effectively defines a variable (existing in filestore) containing four words. The space for these four words form part of the user environment (see 8.1 and 9.). The deref and assign fields of this structure allow one to access and change these four words, subject to certain constraints. The field, spec, is a pointer to filestore disc block produced by some call of npb_to_d; when used by a compiler this will generally be some translation of the interface specification of a compiled module.

The deref field is a disc-pointer to a procedure of mode PROC DATA and where DATA has been produced by the compiling system :

```
MODE DATA = STRUCT( PROG prog,
                    SPEC shaky_spec,
                    EDFILE text,
                    WORD spare)
```

Here the shaky_spec field is a shaky version of a disc-pointer to an interface specification as above. Its shaky properties allow one to recover the space occupied by DATA when the MODULE is no longer referred to.

The assign field of a MODULE is another disc-pointer, this time to a PROC(DATA, BOOL)VOID. Letting these two procedures be deref and assign respectively, the call assign (d, b) will alter the DATA referred to by the MODULE to d if b is TRUE (as in the function change_spec). If b is FALSE and the interface specifications given by specOFd and shaky_spec OF deref are the same, then the prog, text and spare fields the MODULE are changed to that of d (as in the function amend). The comparison of the specs is done by reading the non-pointer blocks given by the specs and comparing them character by character. If b is FALSE and the specs are not the same then the call fails.

6.1.1 PROC new = (PROG prog,SPEC spec,EDFILE text)MODULE:

This procedure creates and initialises a new MODULE in the current environment. The initial DATA in the MODULE is:

(prog, shake_dptr(spec), text, 0)

The three words which form the parameters of new are usually the result of a compiler e.g. the procedure algol68.

6.2 Assembling program

Most progam loading in Flex is done implicitly in the interpretation of Curt commands; what follows here is a description of the primitives involved.

In order to load the program corresponding to a MODULE, one applies a procedure of mode LOADER to it where:

```
MODE LOADER = PROC(MODULE)KEEPS
```

where KEEPS is a (mainstore) pointer to a block containing the set of interface values of the program. The internal structure of this block is deducible from the spec field of the module; its details depend on the compiler and language. A call of a loader, say load(m), will fail in a characteristic manner if specOFm and specOFderefOFm are unequal; one cannot load programs which contain inconsistent use of modules.

The actual loader used depends on context; the user can write his own. The action of a loader is usually fairly trivial, since the prog field of the DATA is itself a disc procedure produced by the compiler which evaluates its own KEEPS and loads any modules used by it. In order to do this, it requires to call the loader recursively on any internal module, and so the procedure in the prog field has a loader as a parameter i.e. the mode PROG is a disc-pointer to:

```
PROC(LOADER,INT)KEEPS
```

The integer parameter is currently not used by the compilers on Flex.

6.2.1 PROC get_module = (MODULE m, LOADER l,INT c)KEEPS

Not all program defined by modules lives in filestore; most of the procedures in the kernel, for example, can only exist in main memory in a transitory fashion. These modules are recognised by this procedure, which gives the required KEEPS to these "in-store" modules as well as doing the standard operations as described above to normal modules. Thus, if m is a standard module then the answer is the answer to a call of the procedure derived from the prog field of m with l and c as parameters; otherwise, the KEEPS corresponding to this module is already bound to the procedure and this is the answer to get_module. A non-standard module is recognised by its deref and assign fields, both being scalars rather than disc-pointers.

The kernel and system procedures defined in this paper are mostly accessible in program using the module names (defined in the common dictionary) formed by using the _m suffix on the procedure name e.g. vdu_m, from_disc_m etc.

A loader is usually constructed by using the standard, procedure make_loader:

```
PROC make_loader = ( PROC(MODULE,LOADER,INT)KEEPS g_m, INT c )LOADER
```

The answer to a call of make_loader is a loader which calls g_m exactly once for each different module found in the tree of evaluation of the program being loaded; the loader supplied as a parameter to g_m is this loader itself. If the same module is encountered more than once then the KEEPS which was the result of the first evaluation is used for subsequent ones. This means that the same set of values are bound to the program, regardless of how many times the module was used.

The reason for this apparent complication in loading and assembling is that the meaning of a particular module may depend on the context of use. As a trivial example, the module vdu_m must give different values for each Virtual Machine. A less trivial example occurs where one enters a new naming regime; the procedure find (which gives a value corresponding to an identifier, see 7.2) contained in the module find_m must allow one to enter (and leave) new scopes. The general method for doing this is to re-define get_module in an inner scope, usually via the command interpreter, Curt. The new get_module would intercept some modules, delivering already evaluated KEEPS in these cases and call the old get_module for the other cases. Notice that the module get_module_m almost certainly would be one of the intercepted modules, delivering the new get_module in its KEEPS, so that a further inner redefinition will include the outer one.

6.2.2 PROC current_loader = (MODULE m)KEEPS

This procedure is the loader which was used to load the currently running program. A call of this procedure from within this program will load the requested module normally if it is not already part of the program; if it is, then the interface values already bound into the program are delivered.

7. Name and value identification

Most name and value correspondences made in Flex are done at the level of the Curt interpreter or within compilers. The following procedures are those primitives used to do it.

The meaning of a name in Flex is dependent on the context or environment of use. There are two different ways of naming an object - first, where the name is only valid for the current session (temporary) and secondly where the meaning of the name persists from session to session. Any value created in Flex can be named in the first way; however the second way can only be done for filestore objects. A filestore object is essentially one which contains no main-store pointers - it can of course contain disc-pointers.

Most users of Flex have access to two dictionaries which gives the file-store name-value relationships; one which is private to the user which he can update and the other which is held in common across the system which he cannot update. Both of these of these are obviously held in file-store. The temporary dictionary is held entirely in main-store and no name clashes are permitted between this and the private file-store dictionary. Thus the temporary dictionary can be regarded as a main-store extension of the private file-store dictionary. Any name clashes between the private and common dictionaries are resolved by giving precedence to the former.

7.1 PROC basic_find = (VECTOR[]CHAR name)REF VECTOR[]UVALUE

```
where UVALUE = UNION(REF VECTOR[]INT,
                     REF VECTOR[]STRUCT(INT fn),
                     REF VECTOR[]CHAR,
                     REF VECTOR[]BOOL )
```

This procedure delivers the set of values of mode UVALUE associated with the name in the current environment. Each of these values can be vectors of integers, characters or booleans. The remaining element of the union is somewhat historical and is intended to differentiate those values which are in fact disc procedures. The first value of the result vector is the principal value. There need not be any others, but if they are given the second is the Curt mode of the principal value, the third a LONG INT representation of the date and time the name was declared and the fourth an editable file which contains information on the value; this last is the file accessed by the function info. Thus, given that d_to_b_m is a name in the common dictionary, the call basic_find("d_to_b_m") would deliver a vector of four UVALUEs, each of which is formally a vector of integers. The first vector would have three elements, since d_to_b_m is actually a MODULE and the use of some suitable unpack operator (opcode 167) on the integer vector could produce this MODULE. The second vector has one element which happens to be a disc pointer, the vector itself being a representation of the Curt mode for Module. The third vector has two elements giving the date and time while the fourth has one which is an editable file containing roughly the same text as given in 3.3.1.

The most common use of basic_find is in the construction of new find procedures as below using the make_find procedure. The procedure make_find takes a user's basic_find as parameter and delivers a new

find procedure, making sure that this will bind the new find, basic_find and get_module into any calls of new environments using this new find as well as any new name associations required by the user.

7.2 PROC find = (VECTOR[]CHAR n)VM:
Where VM = STRUCT(REF VECTOR[]INT value,mode)

A call of this procedure find gives the value and Curt mode corresponding to the string n in the current environment. If n is not defined in this environment then the call fails.

Although the formal Algol68 mode of the value is a vector of integers, the actual value could equally well be a vector of characters or booleans. Thus, a suitable manifestation of the unpack operator (opcode 167) could transform this value to any Algol68 value consisting of a number of words, characters or booleans. Similarly, the use of the pack operator (opcode 166) can produce the vector of integers from any Algol68 value.

The representation of the Curt mode corresponding to the name is defined system-wide. Thus, if the name had corresponded to an editable file then the Curt mode would have been Edfile and its representation is that given by the call make-mode("Edfile").

The find procedure that is bound to a program is usually the same as that used as a parameter to the current call of the command interpreter curt (see 9.1).

7.3 PROC keep = (INT dec_type,VECTOR[]CHAR n,VM v)VOID:

A call of keep adds (or redefines) the name n to correspond to the value-mode pair given by v in the current environment. Thus a call of find in this environment with n as parameter will deliver v. If dec_type = 1, then the association is temporary and will disappear at the end of the current session. If dec_type = 2 then the association is held in file-store and will persist between sessions. In this latter case the value being named must be a file-store value and in the case of a redefinition of an existing name the modes must be the same. No temporary name can be defined if this name is already defined in the persistent sense and vice-versa. If any of these conditions are not met, the call of keep will fail without changing the dictionaries.

The keep procedure that is bound to a program is usually the same as that used as a parameter to the current call of the command interpreter curt (see 9.1).

8. Dictionary utilities

The file-store dictionary belonging to a user is created at the same time as the user is given access to the system by having an environment procedure named in the outermost environment (see 9). Thus the environment for casual non-serious users has an environment procedure called play in the outer environment. At the creation of the dictionary it was initialised to contain three functions which are particular to the environment. The first, new, has already been described (6.1.1) and creates new modules in this environment. The other two are concerned with showing and tidying the file-store dictionary. These are given names specific to the user, using the same name as the environment procedure. Thus these procedures in the play environment are called show_play and tidy_play. In what follows the user name is denoted by User.

8.1 PROC show_User = PAGE:

This procedure will display the file-store dictionary of User using the editor. The PAGE delivered by the procedure (by keying RESULT) is simply the main-store version of an editable file produced by editing the display.

This display consists of two initial lines, followed by a set of line triples, one for each name defined in the dictionary. The initial lines are values of Curt mode Module_set and Old_dictionary respectively. The Module_set is a pointer to the set of modules created in this User environment. The Old_dictionary is a shaky disc pointer to the contents of the dictionary before it was last updated. This value can be used to retrieve old values by using the function show_old provided that a disc garbage collection has not occurred.

The line triples following this define each name held in the dictionary in alphabetical order. The first line contains the name and the time and date on which it was declared (the cursor will appear over the most recent declaration). The second line is the value corresponding to this name while the third, if not empty, is the documentation file associated with this value. Both of these values can be accessed in the normal edit sense.

If one wishes to delete names from the dictionary then one deletes the corresponding line triples from the display and applies tidy_User to the result of the procedure.

8.2 PROC tidy_User = (PAGE p)VOID:

The parameter of this procedure is intended to be derived from show_User (or perhaps from some show_old). The identifiers and values given in the PAGE will become the new dictionary for the User.

8.3 PROC show_temp = PAGE:

This procedure shows the temporary names and corresponding values in the same way as show_User. At the same time it removes all the temporary values from the local dictionary; if one wishes to retain them (or some subset of them) then one uses tidy_temp.

8.4 PROC tidy_temp = (PAGE p)VOID:

The parameter of this procedure is intended to be derived from show_tidy and gives a page of identifier-value correspondences which will form the new temporary dictionary after the call.

9. The User procedure

A User procedure accessible at the outermost level of Flex (e.g. one accessible by name when a vdu is started up) consists of the call of an environment procedure with a dictionary belonging to the user and some other finding procedures already bound in. The dictionary belongs to the user in the sense that only he has the procedures to find and update identifiers in it (he can of course give them to other users if so desired).One of the other finding procedures is one which looks for identifiers in the common dictionary. This includes most of the modules and functions mentioned in this document and also the function show_common which displays its contents. Normal users do not have access to the keeping procedure which updates the common dictionary; in fact, this is contained in the priv dictionary whose finding procedure is bound into some privileged Users in the same way as the finder for common is bound into all Users

The names in a User's dictionaries (both temporary and persistent) as well as all those given by the bound finding procedures form the domain of the find procedure given in 7.2. These names are therefore accessible within program; they are also accessible directly by the command interpreter, curt, which is called in each User procedure. In fact, only those names are accessible in the interpretation of commands and there is no distinction between those created by the user and those already present in the system (in common, say).

The User procedure also maintains a monitoring file of noteable events; these events are usually where the persistent dictionary changes. Entries are made into this file by means of the procedure monitor and it can be accessed by the procedure get_m_file. Previous states of the dictionary can be reinstated by using the procedure unwind on a truncated version of the monitoring file.

In the following description, a typical User is expressed as an Algol68 procedure; usually it would not be called directly within an Algol68 program but only by the command interpreter as part of the interpretation of a command line at the outermost level.

9.1 PROC User = (VM par)VM:

The Curt mode of User is Moded -> Moded

The parameter par is any value-mode pair. The Curt mode rules means that any Curt value can be supplied as parameter in a Curt interpretation of a call of this procedure. This value is declared to have name Op in the initialisation of the temporary dictionary of the environment. Another name in the initialisation is where which is used to give an indication of the context of the environment by the contents of the Curt line each time the environment changes by inner call of Curt (more strictly inner calls of make_find).

The procedure User then continues by searching for a name PASSWORD in the various places available to it. If it finds a corresponding value then it treats this as a PROC VOID and calls it. Most PASSWORDs are created and declared in the user's dictionary by the procedure, password, which binds its string parameter to a procedure which asks one to type it out (invisibly) on the vdu and fails if it does not match (thus also failing the call of User) . Other more complicated

procedures could be invented by the user.
The procedure User then creates all the environment dependant values and procedures (eg find, keep, get_module etc) relevant to this user and calls the procedure, curt.

```
PROC curt =
      ( PROC(VECTOR[]CHAR)VM find,
        PROC(INT,VECTOR[]CHAR,VM)VOID keep,
        PROC(INT,REF VECTOR[]CHAR)VOID monitor,
        VECTOR[]CHAR first_line) VM
```

This procedure interprets commands typed at the vdu and is discussed in [2]. The find parameter is the procedure that the interpreter will use to give meaning to identifiers typed on the command line and its actual in this call is the find procedure of the environment. Similarly the keep parameter allows one to do declarations of identifiers on the command line to update the dictionaries of the environment by using the keep procedure of the environment. The first_line parameter of curt is the first line that will be interpreted and in this call it is the empty string if the mode part of the par parameter of User is Void; otherwise it is the string "Op". The monitor parameter is called each time a declaration is made on the Curt line with its string parameter being the Curt line itself and its integer parameter being 1 for a temporary declaration and 2 otherwise. The actual used is again generally available and is described in 9.2.

The answer delivered by User is the answer delivered by the call of curt.

9.2 PROC monitor = (INT t, REF VECTOR[]CHAR mess)VOID:

If t is not 1, then the line mess is appended to the current monitoring file followed by the current state of the user's persistent dictionary; this last appears, in a mendacious fashion, as an Int in the file. The procedure monitor is called, for example, at each declaration in curt, at every amend module, at every new module and every time a dictionary is tidied; the intention is to note each possible change to the dictionary. Besides giving a record of what one has done, the monitoring file can also be used by the procedure unwind to forget about undesirable changes made to one's dictionary.

9.3 PROC get_m_file = EDFILE:

The result of a call of this procedure is the current state of the monitoring file.

9.4 PROC unwind = (EDFILE mon)VOID:

The parameter of this procedure must be a truncated version of the current monitoring file, ie the monitoring file at some earlier time in the session. This can only be produced by editing the result of a call of get_m_file with the only permitted actions being to delete final pairs of lines (NB the lines are 160 chars long). If this

condition is not met the procedure will fail without altering the dictionary. If it is met then both the dictionary and the current monitoring file is returned to what it was at the earlier time given by the truncated monitoring file.

10. Conclusion

This paper is not intended to be a teaching document but rather a description of the basic building blocks of the Flex software. The easiest way to learn how to use Flex is to use Flex; a good deal of thought and energy has gone into the design of a friendly user interface and on-line teaching and information aids. The procedures described here are mainly those which are not directly used by most users; they tend to be wrapped up in less basic procedures.

There many other procedures in Flex which might have been included in this paper. Noteable ommissions includes the listing and networking procedures. In addition, the editor and the structure of editable files form such an important part of programming on Flex that it could be considered as part of the System in spite of the fact that it does not meet my criteria for inclusion given in the Introduction. However, as each of these topics merit a complete paper to themselves, the set given here probably forms a reasonable compromise between brevity and completeness. This set is sufficient to give all the primitives required for writing compilers, editors, database managers, ... - in fact, the basic stuff of software.

References

[1] Flex Firmware RSRE Technical Note 81009, Currie,Edwards and Foster

[2] Curt: The command interpereter for Flex RSRE Memo 3522, Currie and Foster

11. Index and Glossary

algol68	Algol68 compiler - with Curt mode Edfile->Moded where Moded is either an Edfile (failed compilation) or a Compiledpair(successful). mentioned 6.1.1
amend	function with Curt mode Module -> (Compiledpair ->()) used to update program without changing specification. mentioned 6.1
basic_find	finds set of values associated with an identifier in current environment, 7.1
break_si	soft_interrupt belonging to the break-in process, 2.4
c_to_d	creates a code-block on disc,3.2.3
code-block	that part of a procedure which is constant,[1],3.2.3,6.
common	dictionary containing values generally avaiable across the system, mentioned 7. , 9.
current_loader	loader used to load current program, 6.2.2
curt	command interpreter used by Flex system,[2] and see 9.1
d_to_b	transforms a disc-block into corresponding main-store block, 3.3.1
DATA	Algol68 mode of object refered to by a MODULE,6.1
DISCPTR	mode of a disc pointer , INT in Algol68, 3.
DISCREF	mode of a disc reference, INT in Algol68, 3.
date	today's date ,2.8
diagnose	Curt function used to diagnose run-time errors, see 5.
disc-pointer	held as a pointer to a keyed block in main-store held as recognisable bytes on file-store, 3.1 etc.
disc-reference	disc-pointer which is a disc variable containing one word, 3.1, 3.4.
EDFILE	Algol68 mode corresponding to Curt mode Edfile, describing an editable file, mentioned 8.
Exceptions	Failures either produced by software or firmware,[1],5.
failure	innaccessible Kernel procedure used to construct diagnostic chain at exceptions, see 5.

find	gets the value-mode pair associated with an identifier in current environment,7.2
firm_dptr	firms a shaky disc pointer,3.5.1
from_disc	reads the contents of disc block, 3.3.2
garbage collection	main-store - implemented in the micro code of Flex [1] file-store - done off-line , see 3.1
get_module	delivers the interface values of a module,6.2.1
get_m_file	delivers the current monitoring file, 9.3 Curt mode : () -> Edfile
info	function to get the information file on an identifier; mentioned 7.1. Curt mode : Vec Char -> Edfile
keep	used to declare identifiers in current dictionaries,7.3
LOADER	Algol68 mode of loader ,defined in 6.2
make_find	creates a new find procedure from a basic_find, 7.1,9.1
make_loader	creates a loader,described in 6.2.1
make_mode	function to construct a Curt mode, mentioned 7.2 Curt mode : Vec Char -> Mode
make_process	creates a new process,2.2
make_sema	creates a new semaphore,2.5
Moded	Curt mode corresponding to Algol68 mode VM; in curt interpreter, used to transfer mode information into and out of procedure calls; mentioned 9.1
MODULE	Algol68 mode with corresponding Curt mode Module, 6.1
Module_set	Curt mode for pointer to set of modules, 8.1
monitor	records a message and state in monitoring file,9.2
new	creates a new module, 6.1.1 when used as a Curt function its mode is Compiledpair -> Module (see algol68,amend etc)
new_dr	creates a new disc reference,3.4.2
npb_to_d	write non-pointer data to disc,3.2.1
Old_dictionary	Curt mode for shaky disc pointer to previous state of dictionary, see 8.1

own_si	soft_interrupt for principal process,2.3
own_time	time spent by current process ,2.7
p_to_d	creates a procedure on disc,3.2.4
PAGE	Algol68 mode corresponding to Curt mode Page which is a mainstore representation of an editable file. Mentioned 8.1,8.2,8.3,8.4
PASSWORD	name of password proc (Curt mode ()->()) in User dictionary, usually put there by proc password. see 9.1
password	function (Curt mode (Vec Char)->()) which puts a PASSWORD into User dictionary, see 9.1
pb_to_d	writes words (including disc-pointers) to disc,3.2.2
priv	a dictionary only accessible to certain Users, see 9.
shake_dptr	makes a disc pointer shaky, 3.5.1
show_common	display contents of common dictionary, mentioned 9. Curt mode : () -> Page
show_old	display contents of old state of dictionary, mentioned 8.1. Curt mode : Old_dictionary -> Page
show_temp	display contents of current temporary dictionary ,8.3 Curt mode : () -> Page
show_User	display contents of User's persistent dictionary ,8.1 Curt mode : () -> Page
soft_interrupt	characterises a process, 2.2
SPEC	mode of disc ptr to "specification" of module, see 6.1
tidy_User	usually used to delete names from User's persistent dictionary, 8.2 Curt mode : Page -> ()
tidy_temp	usually used to reinstate names in current temporary dictionary, 8.4 Curt mode : Page -> ()
timed_wait	procedure for waiting a bit,2.6
to_dr	write to disc-reference ,3.4.1
unwind	restore dictionary to earlier state, 9.4 Curt mode : Edfile -> ()
User	name of a typical environment proc, accessed at outer level, 9. Curt mode : Moded -> Moded

UVALUE	**Algol68 mode for a value described in 7.1**
vdu	**procedure for using ones vdu, 4.**
Virtual Machine	**the Flex machine as seen by a single user, see 2.**
VM	**Algol68 mode giving a value-mode pair described in 7.2**

DOCUMENT CONTROL SHEET

Overall security classification of sheet ....UNCLASSIFIED.....

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the box concerned must be marked to indicate the classification eg (R) (C) or (S))

1. DRIC Reference (if known)	2. Originator's Reference	3. Agency Reference	4. Report Security Classification
	Memorandum 3636		u/c
5. Originator's Code (if known)	6. Originator (Corporate Author) Name and Location		
	Royal Signals and Radar Establishment		
5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location		

7. Title

Kernel and System Procedures in Flex

7a. Title in Foreign Language (In the case of translations)

7b. Presented at (for conference papers) Title, place and date of conference

8. Author 1 Surname, initials	9(a) Author 2	9(b) Authors 3,4...	10. Date	pp. ref.
Currie, I F	Foster, J M	Edwards P W		

11. Contract Number	12. Period	13. Project	14. Other Reference

15. Distribution statement

Unlimited

Descriptors (or keywords)

continue on separate piece of paper

Abstract

This Memorandum describes the basic Kernel and System procedures on which the operating system for the Flex computer is based. These are the low level procedures which are used to implement the compilers, file-store, command interpreters etc on Flex.

END

FILMED

1-84

DTIC